Buffer Overflows

Address space layout, the stack discipline, + C's lack of bounds-checking = HUGE PROBLEM

getaddrinfo()
Outline

*Goal*: how the stack + lack of bounds checking make C program vulnerable to a certain (serious!) type of security vulnerability

- Understanding buffer overflows
- Refresher on memory layout
- C library function: `gets`
- `gets + echo` buffer overflow example
- Simplified security exploit example
- Buffer overflows in the wild
- When this is a problem
- Real-world implications
- Unit summary
x86-64 Linux memory layout

Not drawn to scale!
C: String library code

C standard library function `gets()`

```c
/* Get string from stdin */
char* gets(char* dest) {
    int c = getchar();
    char* p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

What could go wrong when using this code?

Same problem in many C library functions:

- `strcpy`: Copies string of arbitrary length
- `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification
C: Vulnerable buffer code usinggets(...)

```c
/* Echo Line */
void echo() {
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

int main() {
    printf("Type a string:");
    echo();
    return 0;
}
```

$ ./bufdemo
Type a string:123
123

$ ./bufdemo
Type a string: 0123456789012345678901234
Segmentation Fault

$ ./bufdemo
Type a string: 012345678901234567890123
012345678901234567890123

These two lines of code introduce a vulnerability!
Vulnerable buffer code using `gets` : disassembled x86

<table>
<thead>
<tr>
<th>echo code</th>
<th>caller code</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000004006cf <code>&lt;echo&gt;</code>:</td>
<td>40068: 48 83 ec 08</td>
</tr>
<tr>
<td>4006cf: 48 83 ec 18</td>
<td>sub $24,%rsp</td>
</tr>
<tr>
<td>4006d3: 48 89 e7</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>4006d6: e8 a5 ff ff ff</td>
<td>callq 400680 <code>&lt;gets&gt;</code></td>
</tr>
<tr>
<td>4006db: 48 89 e7</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>4006de: e8 3d fe ff ff</td>
<td>callq 400520 <code>&lt;puts@plt&gt;</code></td>
</tr>
<tr>
<td>4006e3: 48 83 c4 18</td>
<td>add $24,%rsp</td>
</tr>
<tr>
<td>4006e7: c3</td>
<td>retq</td>
</tr>
<tr>
<td>4006e8: 48 83 ec 08</td>
<td>sub $0x8,%rsp</td>
</tr>
<tr>
<td>4006ec: b8 00 00 00 00</td>
<td>mov $0x0,%eax</td>
</tr>
<tr>
<td>4006f1: e8 d9 ff ff ff</td>
<td>callq 4006cf <code>&lt;echo&gt;</code></td>
</tr>
<tr>
<td><strong>4006f6</strong>: 48 83 c4 08</td>
<td>add $0x8,%rsp</td>
</tr>
<tr>
<td>4006fa: c3</td>
<td>retq</td>
</tr>
</tbody>
</table>
Buffer overflow example: before input

Before call to gets

Stack frame for call_echo

Return Address

20 bytes unused

buf ← %rsp

void echo() {
    char buf[4];
    gets(buf);
    ...
}

call echo:

00 00 00 00 00 40 06 f6

[3] [2] [1] [0]

buf ← %rsp

4006f1: callq 4006cf <echo>
4006f6: add $0x8,%rsp

...
Buffer overflow example: input #1

```
$ ./bufdemo
Type a string: 01234567890123456789012
01234567890123456789012
Overflowed buffer, but did not corrupt state
```
Buffer overflow example: input #2

After call to gets

Stack frame for `call_echo`

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>40</td>
<td>00</td>
<td>34</td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
</tr>
</tbody>
</table>

Return Address

buf ← %rsp

Null Terminator

Overflowed buffer and corrupted return pointer

unix> ./bufdemo
Type a string: 0123456789012345678901234
Segmentation Fault

```
void echo() {
    char buf[4];
    gets(buf);
    ...
}
```

call_echo:

```
        4006f6: add $0x8,%rsp
```

```
        4006f1: callq 4006cf <echo>
```

```
        echo:
            subq $24,%rsp
            movq %rsp,%rdi
            call gets
            ...
```
Buffer overflow example: input #3

Overflowed buffer, corrupted return pointer, but program seems to work?!
Buffer overflow example: input #3

After call to gets

Stack frame for call_echo

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
</tr>
</tbody>
</table>

Return Address

buf ← %rsp

Null Terminator

Some other place in .text

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>400600: mov %rsp,%rbp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400603: mov %rax,%rdx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400606: shr $0x3f,%rdx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40060a: add %rdx,%rax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40060d: sar %rax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400610: jne 400614</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400612: pop %rbp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400613: retq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Works because: “Returns” to unrelated code, despite what the C code had!
Lots of things happen, without modifying critical state
Eventually executes retq back to main
Exploiting buffer overflows

```c
int bar() {
    char buf[64];
    gets(buf);
    ...
    return ...;
}

void foo() {
    bar();
    ...
}
```

Stack after call to `gets()`
- `foo` stack frame
- `bar` stack frame

B (was A)
- Exploit code
- Data written by `gets()`
- Pad
- Return address A

B
Simplified exploit example (no padding)

```c
#include <stdio.h>

void delete_all_files() {
    // ... users shouldn't be able to call this
}

void read_input() {
    char buf[8];
    gets(buf);
}

int main() {
    read_input();
}
```

read_input:
```assembly
401126: subq $8, %rsp
40112a: leaq (%rsp), %rdi
40112f: movl $0, %eax
401134: call gets
401139: addq $24, %rsp
40113d: ret
```

delete_all_files:
```assembly
40003e: call evil
...
```

main:
```assembly
400048: call read_input
40004d: addq $8, %rsp
400051: ret
```
Simplified exploit example (no padding)

read_input:
0x401126: subq $8, %rsp
0x40112a: leaq (%rsp), %rdi
0x40112f: movl $0, %eax
0x401134: call gets
0x401139: addq $24, %rsp
0x40113d: ret

delete_all_files:
0x40003e: call evil

main:
0x400048: call read_input
0x40004d: addq $8, %rsp
0x400051: ret

Update the stack and registers diagram to the state at the red line

Memory

Stack Frames

main

Update the stack and registers diagram to the state at the red line

%rsp

%rip
Simplified exploit example (no padding)

read_input:
401126: subq $8, %rsp
40112a: leaq (%rsp), %rdi
40112f: movl $0, %eax
401134: call gets
401139: addq $24, %rsp
40113d: ret

delete_all_files:
40003e: call evil
...  

main:
...  ...
400048: call read_input
40004d: addq $8, %rsp
400051: ret

Update the stack and registers diagram to the state at the red line
Simplified exploit example (no padding)

**main**: ...
400048: call read_input
40004d: addq $8, %rsp
400051: ret

**read_input**:  
401126: subq $8, %rsp
40112a: leaq (%rsp), %rdi
40112f: movl $0, %eax
401134: call gets
401139: addq $24, %rsp
40113d: ret

**delete_all_files**:  
40003e: call evil ...

Discuss: how long would the user input on standard in need for a buffer overflow attack? What address would we want to appear, and where, to delete all files?
Exploiting buffer overflows: when is this a problem?

We could construct x86 code to mess up our own programs call stack.

But, we trust our own code to not!

The problem: allowing user input (untrusted source) to potentially corrupt the stack.

Combination of: untrusted input, code that does not enforce bounds.

get(input); strcpy(input, ...); scanf(input, ...); etc

Stack after call to gets():

- foo stack frame
- bar stack frame

B (was A)

pad

exploit code

data written by gets()
Exploits in the wild

Buffer overflow bugs allow remote attackers to execute arbitrary code on machines running vulnerable software.

1988: Internet worm

Early versions of the finger server daemon (fingerd) used `gets()` to read the argument sent by the client:

```
finger somebody@cs.wellesley.edu
```

Attack by sending phony argument:

```
finger "exploit-code padding new-return-address"
```

Still happening

"Ghost:" 2015

getaddrinfo()  
Feb. 2016

gethostname()
Heartbleed (2014)

Buffer over-read in OpenSSL
Widely used encryption library (https)
“Heartbeat” packet
Specifies length of message
Server echoes that much back
Library just “trusted” this length
Allowed attackers to read contents of memory anywhere they wanted
~17% of Internet affected
“Catastrophic”
Github, Yahoo,
Stack Overflow, Amazon AWS, ...
Avoiding overrun vulnerabilities

1. Use a memory-safe language (not C)!

2. If you have to use C, use library functions that limit string lengths.
   - `fgets` instead of `gets`
   - `strncpy` instead of `strcpy`
   - Don’t use `scanf` with `%s` conversion specification
     - Use `fgets` to read the string
     - Or use `%ns` where `n` is a suitable integer

Other ideas?
System-level protections

Available in modern OSs/compilers/hardware
(We disabled these for buffer assignment.)

1. Randomize stack base, maybe frame padding

2. Detect stack corruption
   save and check stack "canary" values

3. Non-executable memory segments
   stack, heap, data, ... everything except text
   hardware support

Helpful, not foolproof!
Return-oriented programming, over-reads, etc.
Conclusion of unit: **Hardware-Software Interface (ISA)**

**Lectures**
(building on everything from HW)
- Programming with Memory
- x86 Basics
- x86 Control Flow
- x86 Procedures, Call Stack
- Representing Data Structures
- Buffer Overflows

**Topics**
- C programming: pointers, dereferencing, arrays, cursor-style programming, using malloc
- x86: instruction set architecture, machine code, assembly language, reading/writing x86, basic program translation
- Procedures and the call stack, data layout, security implication

**Labs**
(building on everything from HW)
- 7: Pointers in C
- 8: x86 Assembly
- 9: x86 Stack
- 10: Data structures in memory
- 11: Buffer overflows (less)

**Assignments**
- Pointers
- x86
- Buffer (less)

- Mid-semester exam 2: ISA November 16 (1 week from today)